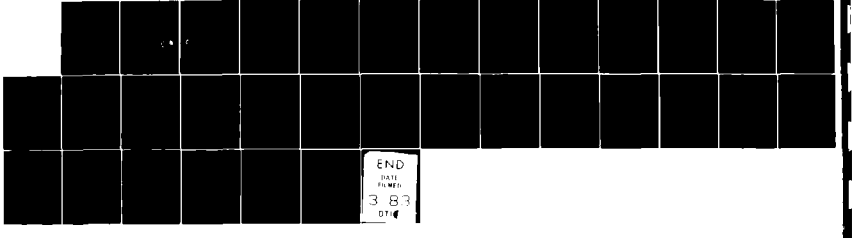
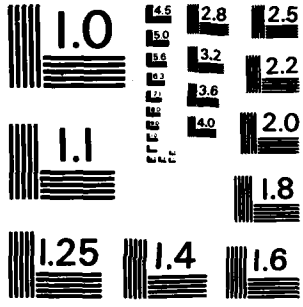
AD A124 261 GPRINT: A LISP PRETTY PRINTER PROVIDING EXTENSIVE USER FORMAT-CONTROL MEC..(U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE L.. R C WATERS 1/1

UNCLASSIFIED SEP 82 AI-M-611A N00014-80-C-0505 F/G 14/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AIM 611A (2)	2. GOVT ACCESSION NO.: AD-A124261
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): GPRINT A Lisp Pretty Printer Providing Extensive User Format-Control Mechanism	5. TYPE OF REPORT & PERIOD COVERED: Memorandum
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Richard C. Waters	8. CONTRACT OR GRANT NUMBER(s): N00014-80-C-0505
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, Massachusetts 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington, Virginia 22209	12. REPORT DATE: Revised Sept. 1982
	13. NUMBER OF PAGES: 27 pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, Virginia 22217	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Distribution is Unlimted

18. SUPPLEMENTARY NOTES

Superseded AD-A109031.

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Pretty Printing
Formatting
Programming environments
LISP

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

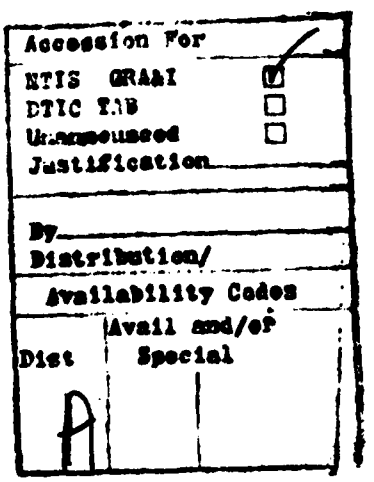
A Lisp pretty printer is presented which makes it easy for a user to control the format of the output produced. The printer can be used as a general mechanism for printing data structures as well as programs. It is divided into two parts: a set of formatted by creating a formatting function for the type. When passed an object of that type, the formatting function creates a sequence of directions which specify how the object should be printed if it can fit on one line and how it should be printed if it must be broken up across multiple lines. con't

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A simple template language makes it easy to specify these directions. Based on the line length available, the output routine decides what structures have to be broken up across multiple lines and produces the actual output following the directions created by the formatting functions. The paper concludes with a discussion of how the pretty printing method presented could be applied to languages other than lisp.



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 611a

Revised - September 1982

GPRINT

A LISP Pretty Printer Providing Extensive User Format-Control Mechanisms

by

Richard C. Waters

ABSTRACT

A Lisp pretty printer is presented which makes it easy for a user to control the format of the output produced. The printer can be used as a general mechanism for printing data structures as well as programs. It is divided into two parts: a set of formatting functions, and an output routine. The user specifies how a particular type of object should be formatted by creating a formatting function for the type. When passed an object of that type, the formatting function creates a sequence of directions which specify how the object should be printed if it can fit on one line and how it should be printed if it must be broken up across multiple lines. A simple template language makes it easy to specify these directions. Based on the line length available, the output routine decides what structures have to be broken up across multiple lines and produces the actual output following the directions created by the formatting functions. The paper concludes with a discussion of how the pretty printing method presented could be applied to languages other than Lisp.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-75-C-0643 and N00014-80-C-0505.

The views and conclusions contained in this paper are those of the author, and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Department of Defense, or the United States Government.

Introduction

Most pretty printers are used solely for formatting program text. They typically operate by reading in a file of program text and producing a formatted text file as output. In general, they have built-in knowledge specifying how each syntactic structure in the programming language should be formatted and do not give the user any significant control over the format of the output produced [1, 2, 4-9]. With such a pretty printer, the lack of user format control mechanisms is tolerable because in most cases the user cannot define any new language constructs and therefore the implementors of the printers can predict in advance all of the structures which the printer can encounter (and though there is no firm consensus on how these structures should be formatted it is possible to select reasonably acceptable formats).

Some pretty printers (such as the Lisp printer presented here) are used as part of the programming environment to display information to the user rather than as text file processors. (Note that an inherent limitation of such printers is that they cannot operate on parts of a program (such as comments) which appear only in text files.) These pretty printers do not have to be relegated solely to printing programs. They can be just as useful for printing data structures. If a pretty printer's use is extended to user defined data structures, user format control mechanisms become essential because it is no longer possible to predict what structures will be encountered.

Extending pretty printers to deal with data is important because user defined data structures are central to almost any program. When debugging a program, a programmer needs to be able to look at various data items. Every interactive programming environment supports the display of the simple atomic data values supported by the language (such as numbers and strings). However, most environments are not prepared to print out the contents of complex user data structures in any useful way.

User defined data abstractions are typically implemented by combining together primitive data structures (e.g. vectors, record structures, and pointers). A pretty printer can be extended to deal with arbitrary user data abstractions by adding print formats for each basic data structure. For example, record structures might be printed as <*field1 field2* . . .> with each field printed on a separate line if the structure cannot be printed on a single line. Vectors could be printed analogously as [*item1 item2* . . .]. Pointers could be printed as '**@**' followed by what they point to. Suppose that a user has defined a data abstraction which is implemented as a record structure with several fields, one of which is a vector of pointers to records. Using the above default formats, an instance of this abstraction would be printed as follows (assuming that several lines had to be used to print it).

```
<field
 field
 [@<field ...>
  @<field ...>
  ...]
 ...>
```

Unfortunately, this simple approach is not very satisfactory. The direct display of the underlying data structure which implements a data abstraction is not liable to capture the user's idea of what the data abstraction means. For example, some components of the data structure may not be very important and should not be displayed at all. Other kinds of data structure components (for example, circular pointers) cannot be displayed literally and must be abbreviated in some way. Alternately, it may be useful to print out some additional quantities which, though not actually in the structure, are useful for understanding the structure (for example, the names of the fields or derived values computed from the field values).

A collateral advantage of the rigid output format initially proposed is that it can be built into the reader as well as the printer so that it is possible to recreate a data structure by reading in its printed representation. In

order to maintain this readability property when fields are being omitted, abbreviated, and/or added in the printed representations for data structures, the user must be careful to insure that no information is actually being lost, and the reader must be modified to take these special printed representations into account. In Lisp programming environments (for example [10]), this kind of reader modification is usually possible though not necessarily easy. It should be noted that in general it is much more important to print out a data structure in a form which can be easily read and understood by the user than to print it out in a form which can be read by the reader.

Another serious problem with the simple output scheme proposed above is that the kind of default formatting rules proposed almost never lead to output which is aesthetic. The visual appearance of a data structure has a very important effect on its understandability. Perhaps different delimiters or indentation would make the data structure more readable. Perhaps the first two fields are closely related and should always be printed on the same line. Perhaps the structure as a whole has two quite separate logical parts which should always be printed on two lines.

In order to deal with these problems, it is essential that the user be able to control how individual data abstractions are to be printed. The pretty printer for Lisp presented in this paper allows the user to specify for each type of data structure both what components to print, and how these components should be formatted. If the printer is used as the standard printer, then the user will be able to inspect his data structures and see them printed out aesthetically at all times.

Pretty printers are typically conceived of as system utilities for displaying information to the user. However, a pretty printer can be much more useful if it can also be used as an output facility which is called directly from user programs. The advantage of this is that it makes available a new paradigm for specifying output format.

Most high level languages have facilities for specifying how output is to be formatted on the page (e.g. the Fortran `FORMAT` statement). In general, these facilities are oriented toward printing data structures whose shape is known in advance on a page whose width is known in advance. There are usually no facilities which deal with variability in either the shape of the data or the width of the page. If either of these has to be parameterized, then the programmer has to write code which computes how each particular data structure should be formatted.

Pretty printers are specifically designed to deal with variability in the data and in the space available. When using a pretty printer, instead of specifying a format for the output as a whole, the programmer specifies individual formats for each of the intermediate structures which can occur in the object to be printed. These formats do not have to be particularly concerned with either the line width or how the intermediate structures will be combined together. When printing a structure, the pretty printer automatically combines the individual formats and decides where to insert line breaks and blank space in order to make its output fit readably in the space available.

The sections below describe how a particular Lisp pretty printer (`GPRINT`) provides for user format control and discuss some of the general issues involved. `GPRINT` was originally implemented in 1975 as an attempt to improve on an earlier pretty printer implemented by Goldstein [3]. Goldstein's pretty printer is one of the few pretty printers which does include mechanisms providing significant user control over the format produced. Unfortunately, the mechanisms he provides are at the same time complex to use and not very powerful. `GPRINT` has been rewritten four times most recently in 1981 in a continuing attempt to create a user controllable pretty printer with very good human engineering.

`GPRINT` is written in Lisp, and was developed in the context of a Lisp programming environment. The Lisp language is used in this paper to display parts of the pretty printing algorithm and Lisp lists are used in examples of how objects are printed. This is done because Lisp has several features which make the

implementation and explication of a pretty printer particularly easy. However, it should be noted that the ideas embodied in GPRINT are not limited to the Lisp domain. In particular, these ideas grow principally out of the requirements for a highly interactive programming environment, rather than out of the Lisp language. The last section of this paper discusses what would be required in order to implement a similar pretty printer for a programming environment other than Lisp.

An Example

Before looking at GPRINT in detail, consider the following example. Suppose a user has defined a data abstraction called NAMED-FORM with four parts: a FORM, which is some arbitrary Lisp expression; a ROOT, which is an identifier associated with the FORM; a SUFFIX, which is used to disambiguate forms which have the same ROOT; and a PARENT, which is a circular pointer pointing up to the NAMED-FORM data structure which contains this one. Together the ROOT and the SUFFIX are a unique name for the FORM. The PARENT links make it possible to go backwards from a NAMED-FORM to the NAMED-FORMs containing it.

The function definitions below implement access functions and a constructor function for this data abstraction implemented as a list. Following common Lisp programming practice, the symbol NAMED-FORM is put in the CAR of this list so that instances of the data type can be recognized at run time.

```
(defun form   (x) (cadr x))
(defun root   (x) (caddr x))
(defun suffix (x) (cadddr x))
(defun parent (x) (car (cddddr x)))

(defun create-named-form (form root suffix parent)
  (list 'named-form form root suffix parent))
```

If nothing more is said, then NAMED-FORMs will be printed out in the default format for lists as follows:

```
(NAMED-FORM (+ A B) ARG 1 ...)
```

There are several problems with this. First, there is no good way to print the circular parent pointer (it is elided as "..." above). Even if some mechanism is used to keep the print form finite, it will probably be too large to be readable. Second, the CAR of the list is important for computational reasons but it is not a logical part of the structure. One might well consider that seeing it printed out is a distraction. Third, the way the remaining three parts of the structure are printed out does nothing to indicate their logical roles in the structure. As a result, it is hard to see what is what.

The following example shows one way in which NAMED-FORMs could be more aesthetically displayed.

```
ARG1: (+ A B)
```

The FORM is printed out preceded by a tag formed by printing the ROOT and SUFFIX as a single unit followed by a colon. Note that you would not want to store the ROOT and the SUFFIX as a single unit because it is computationally expensive to break them apart. However this is easy for your eye to do. The PARENT pointer is not printed at all.

The following format definition could be used to specify to GPRINT that NAMED-FORMs should be printed out in the above way. The expression (DEFUN (*symbol* :GFORMAT) (*arg*) *body*) defines the *body* as a formatting function which will be used to format lists with the indicated *symbol* as their CAR. When passed such a list, the function creates a sequence of formatting instructions specifying what should be printed corresponding to the list. Formatting functions can be quite complex. However, in this example, the formatting function simply selects three of the components of the data structure and calls the function GF (short for GPRINT-FORMAT) in order to create the formatting instructions.

```
(defun (named-form :Gformat) (x)
  (GF "{2 * * ':' - *}" (root x) (suffix x) (form x)))
```

The function (GF *template arg1 arg2 . . .*) creates a sequence of formatting instructions for its arguments based on directions specified by the *template*. (Templates are discussed in detail below.) The template in this example can be understood as follows: The { and } specify that the components between them should be treated as a single logical unit when they are printed out. The 2 after the { specifies that an indentation of 2 should be used inside this structure if it has to be broken up across multiple lines. The three *s show where the three components of the data structure should be printed. The ':' specifies that a colon should be printed after the SUFFIX. Finally, the - specifies a conditional line break. If the whole structure will not fit on one line, then a line break will be inserted at that point. Otherwise a space will be printed.

It is important to realize that the format does not just specify how an individual NAMED-FORM should be printed in isolation. It is used as part of the specification of how complex data structures containing NAMED-FORMs should be printed. For example, a list of two NAMED-FORMs would be printed as follows:

```
(ARG1: (+ A B)
 CALLER3: (- (+ A B) C))
```

The example assumes that in order to fit the structure into the space available for printing, it had to be broken up across two lines. The outermost set of parentheses and the fact that the two NAMED-FORMs are lined up vertically is controlled by the standard format for lists of data. The individual NAMED-FORMs are formatted as specified above.

The Basic Algorithm

The central feature of the algorithm used by GPRINT is that the pretty printing process is divided into two parts as shown in Figure 1. The formatting routine takes in an object and creates a sequence of formatting instructions specifying what to print. These instructions specify how each part of the object is to be printed if it will fit on one line, and how it should be printed if it must be broken up across multiple lines. This information is passed to the output routine as a sequence of entries in a queue. The output routine operates as a coroutine processing the queue entries as they are created. It decides how to fit things into the actual space available and then prints them.

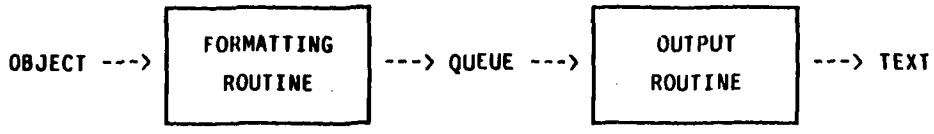


Figure 1: Architecture of the basic pretty printing algorithm.

The importance of dividing the algorithm into two parts comes from the fact that it allows a complete separation between format specification and the output computation. The output routine is complex and computation intensive. Taken separately, it can be designed to be efficient without compromising the need for the formatting process to be as clear and simple as possible. Similarly, when designing the formatting routine and the user format control mechanisms it is possible to concentrate on providing a powerful and convenient interface to the user.

The basic algorithm described above has been independently developed by several people [4, 7] in addition to the author. However, the formatting routines in these other pretty printers are very primitive. They include only a small set of canned formats and do not allow for user format control. In [7], Oppen gives a lucid description of the way the output routine operates. His discussion centers on the fact that if the lookahead used by the output routine when processing queue entries is appropriately limited, then the computation time required by the output routine is linear in the number of queue entries created by the formatting routine. The only difference between his output routine and GPRINT's output routine is that GPRINT's queue entries are more general. Th' paper focuses on the unique aspect of GPRINT -- the way the formatting process allows for user format control.

The Structure of the Formatting Routine

The structure of the formatting routine is based on the idea that any object to be printed by GPRINT can be viewed as a directed graph where each terminal node is a primitive data object (such as a number or a symbol) and each non-terminal node is a composite data structure (such as a list or array). The formatting routine is organized around a central dispatching function (GDISPATCH). At each node, GDISPATCH selects and calls an appropriate formatting function based on various features of the node (such as its data type). The formatting function takes the node as its argument and pushes entries onto the queue which specify what to print and how it should be formatted. Typically, formatting functions call the dispatching function recursively in order to format the composite components of the node.

Consider the following simplified version of GDISPATCH. This version of GDISPATCH assumes that the item to be formatted must be either a number, a symbol, a string or a list. It first tests the data type of the item. If it is not a list then ATOM-FORMAT enters it directly into the queue as something to be printed out. If the item is a list then GDISPATCH looks at the CAR of the list in order to pick a specific formatting function to call. The association between list CARs and formatting functions is recorded by storing the function as the :GFORMAT property of the CAR.

```
(defun Gdispatch (x)
  (cond ((not (listp x)) (atom-format x))
        ((not (symbolp (car x))) (funcall Gnon-symbol-car-format x))
        ((get (car x) ':Gformat)) (funcall (get (car x) ':Gformat) x))
        ((fboundp (car x)) (funcall Gfn-format x))
        (T (funcall Gsymbol-car-format x))))
```

If there is no special formatting function for a list then GDISPATCH uses either a default format for function applications or a formatter for data lists (these formatters are discussed further below). These default formatters are stored in special variables so that they can be easily modified by the user. In a Lisp system there is no definitive way to distinguish the representation of a function call from other kinds of list data. As a heuristic, GDISPATCH looks to see whether the CAR of the list is the name of a currently defined function.

The actual version of GDISPATCH used by GPRINT is much more general than the one presented here. First, it can dispatch on additional features of a list other than its CAR. Second, you can specify a specific format to use when calling GPRINT which will override any dispatching. Third, GDISPATCH dispatches on many other data types as well as lists (for example, arrays). The user format control mechanisms described here are extended so that they are applicable to these other data types. This is discussed in more detail below.

An important thing to keep in mind about formatting functions is that they do not print anything -- rather they specify a set of directions to be followed when GPRINT prints an object of the associated type. In order to print something you call the function GPRINT. It calls GDISPATCH which calls formatting functions which create queue entries which are interpreted by the output routine in order to determine what to print. It is the output routine which actually does the printing.

How The Queue Entries Specify Formatting Options

In order to fully understand how formats are specified, it is important to understand the entries which are pushed onto the queue. These entries are designed to be a concise language for specifying formatting options. The entries encode two pieces of information: what should be printed if an object can be printed on a single line, and where line breaks and indentation should be used if the object will not fit on one line. The following [illegible] describes the basic queue entries.

'*literal*' - Print the literal text between the apostrophes in the output.

_*n* - (Underscore) Print *n* (default 1) spaces in the output. The argument can be negative in which case the printing point moves left but only if there is sufficient blank space to back up over.

{*n* } - These two entries mark the beginning and end of a group of queue entries which form a substructure in the output. This substructure is treated as a single unit when decisions about where to insert line breaks are made. The number following the open bracket specifies how much the indentation should be increased while printing items inside the substructure when they will not fit on a single line. It can be omitted in which case it defaults to the sum of the lengths of the first three things printed in the substructure.

+*n* - (Plus) This specifies a change in indentation. The indentation level in the current substructure is incremented by *n* (default 1) which can be negative.

-*n* - (minus) A conditional line break. Put a line break in the output if the structure immediately containing this entry cannot be printed on a single line. Otherwise, print *n* (default 1) spaces in the output.

! - Always put a line break here.

As an example of how formatting information is encoded in queue entries consider the NAMED-FORM example used above. When GPRINT is used to print the list (NAMED-FORM (+ A B) ARG 1 ...) the formatting routine calls the specially defined formatting function (reproduced below).

```
(defun (named-form :Gformat) (x)
  (GF "{2 * * ':' - *}" (root x) (suffix x) (form x)))
```

Based on the template, the call on GF creates the following queue entries (assuming for simplicity in this example that (+ A B) is formatted as a single atom).

```
{2   'ARG'   '1'   ':'   -   '(+ A B)'   }
```

The output routine processes these queue entries as they are created. It lets the entries corresponding to a structure collect in the queue until it can determine whether or not there is enough room to print the structure on a single line. If the available space is long enough then the entire structure will be printed on a single line as follows:

```
ARG1: (+ A B)
```

If there is not enough room then the structure will be broken up. The - queue entry indicates that in this case a line break should be inserted before (+ A B). The indentation increment specifies that the indentation should be increased by two after the line break.

```
ARG1:
  (+ A B)
```

If there is not enough room to print the two line form, then there is no way to print out the structure which is consistent with the queue entries. This is an example of the *finite line length problem*. Pretty printers in general suffer from this problem and there is no simple solution to it. However, the problem is usually not severe as long as the line length available for printing is several times larger than the largest indivisible item which must be printed on a single line. GPRINT has a number of built-in features (discussed below) which try to ameliorate this problem by keeping the indentation small in order to maximize the line length available.

Formatting Templates

Queue entries are created exclusively through the use of the function (GF *template arg1 arg2...*). GF matches its template against zero or more arguments and produces a series of queue entries. Each template is a string built up out of formatting codes. There are two sets of codes. The first set corresponds exactly to the queue entries described in the last section (i.e. '*literal*', _*n*, {*n* }, +*n*, -*n*, and !). The second set of codes specifies how the template is to be matched against the arguments to be printed. These are described in the table below:

* - Call GDISPATCH to determine how to format this object. If it is an atom then this creates a literal queue entry for it. For example, (GF "*" 'ARG) is the same as (GF "'ARG'").

I - Ignore the corresponding object.

[*subtemplate*] - The part of the object being formatted which corresponds to this part of the template must be a list. It is decomposed into its elements. The template between the square brackets specifies how these are to be formatted. For example, (GF "[*_[*_*]]" '(1 (2 3))) is the same as (GF "*_*_*" 1 2 3). Processing of a subtemplate between [] terminates immediately as soon as the corresponding list is exhausted. For example, (GF "[*':'*]" '(1)) is the same as (GF "*" 1) and not (GF "*':'" 1). The [] codes have meaning only to GF and do not by themselves create any queue entries.

. - (Period) Valid only inside []. It specifies that the next item is the whole sublist left to process by [] rather than its CAR. For example, (GF "[*_.*]" '(1 2)) is the same as (GF "*_*" 1 '(2)).

< > - This is used inside of [] to specify a template for a list of unknown length. The part of the template between the angle brackets is taken as repeating indefinitely, creating a subpattern of infinite length. For example, "[<*_>]" is the same as "[*_*_*_*_*_*_*_ ...]".

(*n subtemplate*) - This is an abbreviation for {*n* '('[*subtemplate*]')'}. This combines together three ideas. First, it specifies that the list should be treated as a single structure in the output. Second, it specifies that parentheses should be printed as delimiters around the list. Third, it specifies that the list should be decomposed using the subtemplate to specify how its components should be formatted. This format code is a useful abbreviation because many list formats share these ideas.

The number after the open parenthesis specifies the indentation increment to use in the substructure. It can be omitted in which case it defaults to the sum of the lengths of the first three entries in the substructure. In this case the first entry is always an open parenthesis. Typically the second entry will be the first item in the list and the third one will be some amount of blank space after the first item.

\# - This can be used in place of an argument to any formatting code (e.g. _, {}, (), +, or -). It specifies that the value is to be taken from the next input to GF. For example, (GF "'A'_#'B'" 6) specifies that 6 spaces should be printed out between the A and the B.

blank - White space can be inserted into a template to give it added readability. It has no meaning in the template.

Consider again the simple template ("{2 * * ':' ~ *}") used in the examples above. The three *s match against the three arguments to GF causing GDISPATCH to be called on each one in turn. The rest of the format codes directly specify queue entries.

Simple Formatting Functions

This section continues the presentation of formatting templates by discussing several standard Lisp program formats. In GPRINT the user format control mechanisms are used to specify all of the standard program formats. This adds greatly to the clarity of the pretty printing algorithm by separating the format specification from the rest of the algorithm. It also makes it possible for the user to modify the way programs are printed by changing the standard formats. It should be noted that in Lisp, programs are represented as lists and are treated just like any other data object. All the mechanisms which allow the user to control the format of program lists can be used to control the format of data structures implemented as lists.

Lisp function applications are traditionally formatted so that they are printed on a single line or, if there is not enough room, so that the arguments are lined up vertically one to a line. The following function is used as the default value of the variable GFN-FORMAT which controls how function applications are formatted. The example printout shows how a function application looks when it has to be printed on more than one line.

```
(defun :Gfn-format (x) (GF "(*_ <*->)" x))

(LIST Y
      Z)
```

The template matches against the list as a whole, printing parentheses around it in the output. The indentation increment is left unspecified so that it will default to the length of the function name plus two (one for the open parenthesis and one for the space printed after the function name). This causes the arguments to line up one under the other. After the function name is printed out followed by a space, the repetitive portion of the template specifies a conditional line break after each argument in the function application. Note that GDISPATCH is called (via the * format code) in order to determine how to format each argument.

Lisp assignments are typically formatted so that each successive variable/value pair appears on a separate line. This can be specified by using the I format code in a template as shown. The following DEFUN sets up a formatting function which specifies that this format should be used for lists which begin with the atom SETQ.

```
(defun (setq :Gformat) (x) (GF "(*_ <*_*I>)" x))

(SETQ Y 1
      Z 2)
```

This template is very similar to the one for function applications. The only difference is that the repeating portion of the template specifies that the arguments are to be formatted in pairs with a mandatory line break after each pair. This forces each pair to appear on a separate line even when the entire SETQ could fit on a single line. Note that there is no line break before the close parenthesis after the last pair because processing in a subtemplate for a list stops immediately as soon as the elements of the list are exhausted.

The LET construct is used to bind a group of variables to initial values and then execute a sequence of statements in this environment. Typically, the variable binding pairs are printed one to a line and the statements are printed one to a line. A small indentation is used for the statements in order to visually differentiate them from the bound variable pairs and in order to keep the total indentation small.

```
(defun (let :Gformat) (x) (GF "(2 *_ (1 <*!>) <-*>)" x))

(LET ((Y 1)
      (Z 2))
  (CONS Y Z))
```

The template specifies an explicit indentation of 2 for the statements in the LET. After the atom LET itself is printed out, a subtemplate specifies how the list of bound variable pairs should be formatted. Here an explicit indentation of 1 is used so that they will line up one under the other. A ! format code is used to force each one to appear on a separate line. The final repetitive portion of the template as a whole specifies a conditional line break before each statement in the LET. Note that if there is only one bound variable pair this allows the let as a whole to be printed on a single line if it will fit.

Conditional expressions are formatted so that each clause of the conditional appears on a separate line. Each clause is composed of a predicate followed by a sequence of statements. If a clause will not fit on a single line, the predicate and statements are printed out one under the other.

```
(defun (cond :Gformat) (x) (GF "(*_ < (1 <*->) ! > )" x))

(COND ((MINUSP Y)
       (- Y))
      (T Y))
```

In this template the repetitive portion of the template as a whole consists of a subtemplate for the clauses and a ! format code which forces each clause onto a separate line. The subtemplate specifies an explicit indentation of 1 and a conditional line break after each expression in the clause.

The following formatting function for MULTIPLE-VALUE-BIND illustrates the use of the + format code. In order to highlight the difference between them, the form which returns the multiple values is printed at an indentation of 4 while the statements which use the bound values are printed at an indentation of 2. The indentation is initially specified as 4. The subtemplate then prints out the list of bound variables. After the multiple value returning form is printed the indentation is decremented by 2. The repetitive portion of the template then prints out the remaining forms one to a line at an indentation of 2.

```
(defun (multiple-value-bind :Gformat) (x) (GF "(4*_ (<*_>) -* +-2 <-*>)" x))

(MULTIPLE-VALUE-BIND (SYMBOL ALREADY-THERE-P)
    (INTERN STRING)
  (COND (ALREADY-THERE-P (ERROR "Symbol already there: " STRING)))
  SYMBOL)
```

As a final example, consider the function QUOTE. A list which begins with the atom QUOTE is not printed with parentheses around it. Rather, the argument to QUOTE is printed out following a "'". The example shows the way the list (QUOTE A) is formatted.

```
(defun (quote :Gformat) (x) (GF "{''''[I *]}" x))

'A
```

The template sets up a substructure and prints a "'" (inside of a literal in a template, "''" stands for "'"). It then prints out the argument to QUOTE. Note how it uses the format codes [] and I in order to select out this argument.

More Complex Formatting Functions

A wide variety of formats can be specified using simple formatting functions like those above which contain only a single call on the function GF. However, these formats are restricted in several ways. In particular, with these simple formatting functions it is not possible to vary the format based on the actual data values in a structure. More complex formats can be specified by taking advantage of the fact that a formatting function can contain arbitrary computation.

For example, consider the following way in which the format for NAMED-FORMs could be extended. Suppose that the suffix field in a NAMED-FORM is optional and that a value of NIL indicates that there is no suffix. In this case we do not want to print the suffix at all. The example shows how the list (NAMED-FORM (+ A B) ARG NIL ...) should be printed.

```
(defun (named-form :Gformat) (x)
  (GF "{2 *" (root x))
  (cond ((not (null (suffix x))) (GF "*" (suffix x))))
  (GF "':'-*}" (form x)))

ARG: (+ A B)
```

In the above format definition the single template used in the format definition in the beginning of this paper is broken into three pieces. A conditional test is inserted so that printing of the suffix only occurs when it is non-null. The { and } indicating the beginning and end of the substructure of queue entries being created are specified in separate calls on GF. This is a common occurrence and is in contrast to [] (and therefore ()) which must be properly nested in a single call on GF.

Of all of the formats in this paper, this is perhaps the best example of the way GPRINT is typically used. Some simple templates are combined with some simple computation in order to define a flexible and aesthetic format for a data object.

Block Form and Tabular Form

In order to save space, long lists of data are often formatted in *block* form where as many items as possible are put on each line. The language which is used to create formatting templates has two format codes which are useful for specifying this kind of format.

,*n* - (Comma) A line break is inserted here if and only if the structure immediately following this code will not fit on the end of the current line. Otherwise *n* (default 1) spaces are printed.

;*n* - (Semicolon) This is the same as the comma format except that additional spacing is inserted so that the items printed out line up in a tabular fashion. The argument *n* specifies what spacing to use between the columns in the table. If it is omitted a default value will be chosen by the output routine based on the lengths of the items to be printed out.

The following formatting function can be used to print out a list in block form.

```
(defun :G1block (x) (GF "(1 <*,>)" x))

(ORANGE PEAR (RED APPLE) GRAPEFRUIT
 (HAWAIIAN PINEAPPLE) BANANA
 CANTALOUPE POMEGRANATE TANGERINE)
```

There is a problem with printing lists of data in block format. If the elements of a list are themselves lists with a depth of greater than one, then the output is not very aesthetic because it is not easy to identify the elements of the top level list. For example, consider the following list:

```
((ORANGE (SELL 3)) (PEAR (BUY 10)) ((RED APPLE) (BUY 5))
 (GRAPEFRUIT (BUY 10)) ((HAWAIIAN PINEAPPLE) (SELL 8))
 (BANANA (SELL 5)) (CANTALOUPE (BUY 4)))
```

The following formatting function uses the semicolon format code in order to print out lists in a tabular format. It is used as the default value of the special variables GSYMBOL-CAR-FORMAT and GNON-SYMBOL-CAR-FORMAT which control how lists of data are printed. This makes the output much easier to read without taking up very much more space.

```
(defun :G1Tblock (x) (GF "(1 <*;>)" x))

((ORANGE (SELL 3))              (PEAR (BUY 10))
 ((RED APPLE) (BUY 5))          (GRAPEFRUIT (BUY 10))
 ((HAWAIIAN PINEAPPLE) (SELL 8))
 (BANANA (SELL 5))              (CANTALOUPE (BUY 4)))
```

Due to the fact that the output routine uses only limited look ahead, the tab size must usually be chosen before all of the elements in the list have been entered in the queue. As a result, it is not guaranteed to be large enough. In this example, the fourth element in the list was not completely entered in the queue at the time when it was determined that the list had to be put on more than one line. As a result, only the first three elements were used to determine the tab size which turned out to be too small to accommodate the fifth element.

Functional Subtemplates

The following format codes increase the flexibility of the templates by making it possible to call functions at different points in a template.

%*f* - This specifies that the function *f* should be called in order to format the corresponding item. The end of the function name is delimited by a space.

$*f* - (Dollar sign) This command specifies that GDISPATCH should be called in order to format the corresponding item, but that the function *f* should be passed to GDISPATCH as a suggestion of how to format the item. As above, the end of the function name is delimited by a space. The difference between **$*f*** and **%*f*** is that with **$*f*** GDISPATCH gets control. As a result, if the item is not a list, then the function *f* will not get used.

The use of the **$** code is illustrated in the following format which block formats a tree at all levels. It is capable of formatting trees of arbitrary depth because it explicitly calls itself recursively. GDISPATCH is called at each level of the recursion. As a result, as soon as an atom is encountered, the recursion is terminated and the atom is printed normally.

```
(defun :Gblock (tree) (GF "(1<$:Gblock ,>)" tree))

(ONE (TWO THREE)
 ((FOUR FIVE) SIX
  SEVEN)
 EIGHT NINE)
```

The following formatting function for PROG uses **%** so that it can call a subformat (GPROG-FORMAT2) *without* GDISPATCH being called. This is necessary so that the labels (which are atoms) in the PROG will be processed by GPROG-FORMAT2. Labels are printed left shifted by computing negative arguments for _.

```
(declare (special Gwas-label))

(defun (prog :Gformat) (list)
  (let (Gwas-label)
    (GF "(*_$:Gblock <%Gprog-format2 >)" list)))

(defun Gprog-format2 (item)
  (cond ((not Gwas-label) (GF "!")))
  (cond ((atom item) (setq Gwas-label T)
         (GF "_#*_" (- (1+ (flatsize item))) item))
        (T (GF "*" item) (setq Gwas-label nil))))

(PROG (RESULT)
    L (COND ((NULL LIST) (GO THE-END)))
      (SETQ RESULT (CONS (CAR LIST) RESULT))
      (SETQ LIST (CDR LIST))
      (GO L)
THE-END (SETQ RESULT (NREVERSE RESULT))
      (RETURN RESULT))
```

An important aspect of the last example is the way it interacts with length abbreviation (described below) and other standard facilities provided by GPRINT. Since length abbreviation is implemented by **[]**, in order to get length abbreviation to apply to the formats you write, you have to use **[]**. This is an important reason for writing it in the form given above rather than as a single routine containing a loop which decomposes the list itself and creates the correct format codes.

Miser Mode

GPRINT provides several facilities which help deal with the finite line length problem. The most comprehensive of these is a modified form of the miser mode supported by Goldstein's pretty printer [3]. The point at which miser mode is triggered is controlled by the variable MISER-WIDTH (which defaults to 40). If the line width available for printing is less than MISER-WIDTH, then miser mode is triggered, and formatting is modified in two ways. First, all indentations inside {} formats are forced to be 1 no matter what is specified. Second, all + formats are ignored so that the indentation remains 1 in each substructure. In addition to this, a formatting command (M) is provided so that the user can specify line breaks which should only happen when miser mode is triggered.

M - A line break is inserted here if and only if the containing structure cannot be printed on one line, and the width available for printing is less than MISER-WIDTH.

~*n* - (Tilde) Print *n* (default 1) spaces in the output. The argument can be negative in which case the printing point moves left if there is sufficient blank space to back up over.

_*n* - (Underscore) This is actually an abbreviation for ~*n*M. It therefore specifies a miser mode line break.

In order to see how miser mode works, consider the format for MULTIPLE-VALUE-BIND reproduced below. The example shows the format which this specifies in miser mode. The indentation increment is reduced to a constant 1, and the occurrences of _ lead to line breaks when misering. The same effects can be seen in the COND.

```
(defun (multiple-value-bind :Gformat) (x) (GF "(4*_ (<*_>) -* +-2 <-*>)" x))

(MULTIPLE-VALUE-BIND
 (SYMBOL ALREADY-THERE-P)
 (INTERN STRING)
 (COND
  (ALREADY-THERE-P
   (ERROR
    "Symbol already there: "
    STRING)))
 SYMBOL)
```

In order to maintain some of the indentation pattern of MULTIPLE-VALUE-BIND in miser mode, the ~ format code could be used in place of _ and + as shown below.

```
(defun (multiple-value-bind :Gformat) (x) (GF "(2*~ (<*_>) - ~2* <-*>)" x))

(MULTIPLE-VALUE-BIND (SYMBOL
                      ALREADY-THERE-P)
   (INTERN STRING)
 (COND
  (ALREADY-THERE-P
   (ERROR
    "Symbol already there: "
    STRING)))
 SYMBOL)
```

Through judicious choice of when to use ~ instead of _ or +, the user can gain considerable control over how a format will look in miser mode. However, as can be seen above, miser mode is not particularly aesthetic no matter what you do. It exists solely as an emergency measure to prevent printout from overrunning the right margin.

Left Shifting of Major Units

Another way in which GPRINT deals with the finite line length problem is to take logical units of program text (such as LETs, PROGs, and DOs) and shift them left in order to increase the amount of line width available. This process is triggered when the line width available for printing is less than MAJOR-WIDTH (which defaults to 40). Left shifting is illustrated in the example below. The radical reduction in indentation is very effective at increasing the width available. Unfortunately, the nonstandard format reduces readability. This problem is ameliorated by the fact that an entire logical unit is being left shifted, not some arbitrary part of the program.

```
(defun (let :Gformat) (list)
  (Gcheck-indentation list
    #'(lambda (x) (GF "(2 *_(1 <*!>)<-*>)" x))))

(defun Gcheck-indentation (list format-fn)
  (let ((ind (Gestimate-indent)))
    (cond ((> (- Glinelen ind) major-width) (GF "%#" list format-fn))
          (T (GF "!~#';----------'~#'|'!" (- ind) (- ind 11.))
             (GF "~#%#" (- 5 ind) list format-fn)
             (GF "!~#';----------'~#'|'!" (- ind) (- ind 11.))))))

(DEFUN ROOTS-OF-QUADRATIC (A B C)
  (COND ((NOT (ZEROP A))
         (LET ((DISCRIMINANT (- (* B B) (* 4 A C))))
           (COND ((PLUSP DISCRIMINANT)
;--------           |
     (LET ((TERM1 (- B))
           (TERM2 (SQRT DISCRIMINANT))
           (TERM3 (* 2 A)))
       (LIST (// (+ TERM1 TERM2) TERM3)
             (// (- TERM1 TERM2) TERM3)))
;--------           |
                   ))))))
```

Left shifting is implemented by the formatting function GCHECK-INDENTATION. The use of this function is illustrated by the formatting function for LET shown above. It calls GCHECK-INDENTATION passing it the simple formatting function for LET which was described in the beginning of this paper. GCHECK-INDENTATION calls the function GESTIMATE-INDENTATION which looks at the queue of formatting commands and determines what indentation will be used when printing out the LET. Note that this must be computed from the queue because there may be many entries in the queue which have not yet been printed.

If the width available for printing is greater than MAJOR-WIDTH then GCHECK-INDENTATION just calls the formatting function passed to it. (Note that if the & format code was used instead of %, GDISPATCH would think that it was encountering a second (circular) reference to the list being printed and abbreviate it as described in the next section). If the width available is less than MAJOR-WIDTH then GCHECK-INDENTATION spaces back to column zero and prints a comment line which indicates that left shifting is occurring using a "|" to show the indentation which otherwise would have been used. On the next line, the format spaces back to column 5 and calls the formatting function passed to it in order to format the list being printed. Finally, it prints another comment line. Note that the templates make heavy use of the # format code so that the function can compute the appropriate negative spacing.

Abbreviation

GPRINT provides several different abbreviation mechanisms. First, there is abbreviation based on PRINLEVEL and PRINLENGTH as in the standard printer. A "**" is printed for structures which are too deep, and "..." is printed in place of the ends of lists which are too long. The following example shows how the list (1 (2 (3 (4))) A B C) would appear with PRINLEVEL and PRINLENGTH both set to 3.

```
(1 (2 (3 **)) A ...)
```

There is a separate abbreviation facility based on the variables PRINSTARTLINE and PRINENDLINE. As GPRINT prints, it counts the lines starting with zero for the line the printer is called on. While the line number is less than PRINSTARTLINE no actual printing is done. If the line number ever becomes greater than PRINENDLINE, then the printer prints "---" to indicate that truncation has occurred and immediately stops printing and returns normally. Experimentation has shown that setting PRINENDLINE to a relatively small number like 4 (while setting PRINLEVEL and PRINLENGTH to NIL) is very useful particularly due to the availability of the continuation facilities described below. The example below shows how an example of output using these settings.

```
(DEFUN ROOTS-OF-QUADRATIC (A B C)
  (COND ((NOT (ZEROP A))
         (LET ((DISCRIMINANT (- (* B B) (* 4 A C))))
           (COND ((PLUSP DISCRIMINANT) ---
```

Truncation of the output can also be triggered by typing TERMINAL STOP-OUTPUT. This interrupts the *printer immediately, causing it to terminate returning* normally.

Whenever output is abbreviated due to any of the methods described above, GPRINT remembers the state of the printing so that it can be resumed. Only a single variable is maintained so that only the most recently abbreviated thing is remembered. If printing was truncated by PRINENDLINE or user intervention, then it can be continued from the point of truncation by typing TERMINAL RESUME.

As an additional feature, you can reprint the last abbreviated thing in full with PRINLEVEL, PRINLENGTH, PRINSTARTLINE, and PRINENDLINE abbreviation disabled by typing TERMINAL 1 RESUME.

As a third kind of abbreviation, if the variable GCHECKRECURSION is T then GPRINT checks for circularity in the objects it is printing. When a circular reference to an object is encountered, it is replaced in the output by ^*n* or %*n*. %*n* is only used in a list. It is used when the CDR of a list is EQ to an earlier CDR in the *same* list. In this case *n* is the number of CDRs separating the two positions. ^*n* is used in other situations. Here, *n* indicates that *n* selector operations (CAR, CXR, AREF; but not CDR) were performed between the first occurrence of the object and this one. This kind of abbreviation is illustrated below.

```
the result of (LET ((X '(Y (Z 1 2 3) 4)))
                 (RPLACD (CDR X) (CDR X))
                 (RPLACA (CDADR X) X)
                 (RPLACA (CDDADR X) (CADR X))
                 (RPLACD (CDDADR X) (CDADR X))
                 X)

prints as     (Y (Z ^2 ^1 . %2) . %1)
```

It is possible (but not easy) to reconstruct the exact shape of the object from what was printed. However, the main purpose is just to print something more readable than what you would otherwise see. An important feature of the way this abbreviation is done is that it is completely orthogonal to the rest of the formatting process so that it works no matter what kinds of user formatting functions are written, and no matter what kind of data objects are being printed.

Data Types Other than Lists

In addition to lists, GPRINT has built in formatters for all of the standard Lisp data types. Symbols, numbers, strings, and things of random types not specifically discussed below are treated as indivisible atoms and printed in the standard ways.

Named structures, entities, and instances are printed in one of two ways depending on whether or not they know how to format themselves. If the object accepts the message :GFORMAT-SELF then GPRINT sends a :GFORMAT-SELF message with the object as argument to the object so that it can format itself.

If the named structure, entity, or instance does not take a :GFORMAT-SELF message, then GPRINT treats it as an atomic object and lets the standard printer print it. This makes it possible to use GPRINT on these objects without having to write formatters for them. However it should be noted that since they are treated as atomic objects, no formatting occurs inside them no matter how large their print form may be. For example, a line break will never be inserted inside one.

If an object is an array (and not a named-structure) it is formatted as follows. GPRINT first checks to see if there is a formatting function for the array. The association between formatting functions and arrays is maintained through a list of functions stored in the variable GARRAY-FORMATTERS. These functions are just like the formatting functions described above except that in addition to creating queue entries in order to format an object, they must also test to see whether they are applicable to the object. This makes it possible for the user to use any kind of applicability test he desires. If the format function is applicable it should format the object and return T. Otherwise it should take no action and return NIL. A function is set up as an array formatter by adding it to the list GARRAY-FORMATTERS. GPRINT calls each of these functions in turn passing it the object. As soon as one of them returns T it stops. If they all return NIL then a default formatter is used.

The default array formatter first prints out the array object in the standard way (e.g. as an atom containing the type and the address). Next, if the variable GPRINT-ARRAY-CONTENTS is T and the array has only one or two dimensions it prints out the contents of the array. The contents are printed as a list (for one dimensional arrays) or a list of lists (for two dimensional ones). Tabular blocking is used to format these lists.

The kind of arbitrary user specified dispatching supported for arrays is also supported for lists. Functions put on the list GLIST-FORMATTERS can be used to associate formats with lists when the association is based on some feature other than the CAR of the list. Similarly, functions put on the list GSPECIAL-FORMATTERS can be used to override all standard dispatching including the initial split based on data type.

Applicability to Languages Other Than Lisp

It is important to note that, though the discussion above was cast in the domain of the Lisp language, the ideas are substantially programming language independent. It should be possible to use these ideas to construct a flexible pretty printer allowing significant user control of format in any programming language environment.

GPRINT makes it possible for the user to control the format of both programs and data. Of these two capabilities, the control over program format is the easiest to export to other language environments. Two basic things are required: a representation for program parse trees, and a method whereby the user can specify formats for non-terminal nodes in these trees. In languages like Lisp where a data representation for parse trees is part of the definition of the language, this is the logical choice for the representation. In other languages some such representation has to be developed. If the pretty printer is intended to accept program text files as input, a parser for the language has to be implemented if one is not already available.

There are two basic ways in which user format control can be supplied. One way is to use the same

mechanisms which are supplied for specifying data formats by simply applying them to the data representations for parse trees. This is the approach taken by GPRINT. Another approach is to follow the suggestion of Oppen [7] and allow the user to specify formats as annotations to the grammar for the programming language. From the point of view of implementation, this approach is essentially identical. However, for a language which (unlike Lisp) has extensive syntax this approach would undoubtedly be aesthetically superior since it uses standard grammatical notation in order to communicate with the user instead of some ad hoc internal representation.

Using GPRINT's approach to the printing of data in other programming environments is more difficult. The key issue is being able to obtain data type information at print time. However, before looking at this problem in detail consider some other issues.

The formatting templates described above could be used with any kind of data. The only thing which has to be changed is that [] has to be extended so that it can decompose other composite data structures besides lists. Logically there is no problem since, in general, any data structure has a default linear ordering for its components. From an implementation standpoint, there is no problem with selecting out components one at a time as long as you can determine the data type of a given structure.

The basic dispatching scheme presented above can be straightforwardly extended as long as type information can be obtained. It is easy to implement an association between types and formatting routines so that each type could have its own format. Further dispatching on subfeatures of individual types could be implemented if desired.

In a language environment such as Lisp where, in general, complete run time type information is available, it is trivial to determine the type of something when it needs to be printed. Unfortunately, in most languages, much of the data type information is used only by the compiler and is not available at run time. In a language with pure strong typing that makes it possible for the compiler to determine the exact data type of every variable, the compiler could be straightforwardly modified in order supply the type information needed by the dispatcher. One way to do this would be to have the compiler create a table of type information which could be referred to by the dispatcher at run time. Alternately, the dispatching needed for individual calls on the printer could be performed at compile time using the compile time type information. In order to make it possible for the user to interactively request the printout of various data items at run time, the tabular approach would be required, just as a dynamic debugger has to have access to the compiler's symbol table in order to use the programmer's variable names.

Unfortunately, few languages have pure strong typing. Most languages support data types such as union types and variant records. Most of the time, this need not be a severe problem because such types are not useful unless there is some way for programs to determine what the actual type of a data item is. For example, the compiler could specify to the dispatcher that a given data item was of a particular union type. The programmer would have to supply a decision procedure which could be used by the dispatcher to determine the exact type of the data item at run time. This would not be a difficult task as long as the union type was straightforward and a single decision procedure for the union type could be implemented which would work in all situations.

There are language environments (for example assembler language) which have little run time type information, little compile time type constraints, and where the user defined data structures are often of such a chaotic nature that it would be virtually impossible to write the kind of data type decision procedures needed by the dispatcher. In such a situation, the kind of pretty printer presented in this paper would not be practical. It should be noted that such an uncontrolled environment presents a number of problems much more serious than the inapplicability of this kind of pretty printing. Current trends have been toward more regularized environments which should be able to support a pretty printer like GPRINT.

Conclusion

GPRINT includes a large number of standard formats and features (such as the ones used as examples above). As a result, a user does not have to write any of his own formats in order to get reasonable output in ordinary situations. However, no amount of anticipation can satisfy every user. This is particularly true when a pretty printer is being used in an interactive programming environment to print data as well as programs, and when it is called by user programs as well as by the system itself.

The principal goal of the design of GPRINT has been to produce a system with good human engineering which gives the user powerful facilities for controlling the format of output and which at the same time makes the specification of simple formats simple. Two key ideas comprise GPRINT's approach to this problem: the basic algorithm chosen, and the existence of multiple levels at which a user can specify formatting information.

The key features of the algorithm underlie the basic simplicity of GPRINT's approach and, at the same time, fundamentally limit its scope. The division of the algorithm into two pieces communicating through a queue makes it possible to separate the simple parts of the algorithm from the complex ones. The decision to use a linear time algorithm in the output routine makes it possible for GPRINT to run with acceptable speed. However, it fundamentally limits the kind of formatting decisions which can be made by the output routine. In particular, when making its decisions, it can only look ahead a very limited distance. An example of this was discussed in the section on tabular form output.

In line with the limited abilities of the output routine the queue entries are designed so that they encode essentially only two formatting options for a given structure: how to print it on one line, and how to print it on multiple lines. (A third miser format is also specified for each structure, however, this format is largely implicit and the user does not have very much control over it.) This design is an important basis for the understandability of the printer because it presents the user with a simple model of how formatting decisions are made. However, one could easily imagine wanting to specify more complex formatting information. For example, one might want to specify two completely different multi-line formats: one to use when there is a lot of room available and the other to use when there is only a little space.

The printer provides three basic levels at which a user can specify formatting information. First, he can simply use the default formats supplied with the printer and does not have to do anything himself. Second, he can use simple templates. These make it very easy for him to describe certain aspects of how a structure is to be formatted. Third, he can write more complex formatting functions. This allows him to exercise much more control over the format to be used, at the cost of greater complexity.

The use of multiple levels of interaction is a generally useful technique for increasing the understandability and availability of a system to a wide range of users. It makes it possible for users who have simple needs to satisfy them without having to learn very much about the system. Users who take the time to learn more can then do more.

References

[1] Conrow, K., and Smith, R.G., "NEATER2: A PL/1 Source Program Reformatter", CACM V13 #11, November 1970, 669-675.

[2] Donzeau-Gouge, V. et al, "A Structure-Oriented Program Editor; A First Step Towards Computer Assisted Programming", Proc. Inter. Computing Symp., Antibes, 1975.

[3] Goldstein, I., "Pretty Printing, Converting List to Linear Structure", MIT/AI/MEMO-279, February 1973.

[4] Hearn, A.C. and Norman, A.C., "A One-Pass Pretty Printer", Report UUCS-79-112, Univ. of Utah, Salt Lake City, Utah, 1979.

[5] Heuras, J., and Ledgard, H., "An Automatic Formatting Program for Pascal", SIGPLAN Notices V12 #7, July 1977, 82-84.

[6] McKeeman, W. "Algorithm 268, Algol-60 Reference Language Editor [R2]", CACM V8 #11, November 1965, 667-669.

[7] Oppen, D., "Prettyprinting", ACM Transactions on Programming Languages and Systems, V2 #4, October 1980, 465-483.

[8] Scowen, R. et al, "SOAP - A Program Which Documents and Edits Algol60 Programs", Comput. J. V14 #2, 1971, 133-135.

[9] Teitlebaum, T., "The Cornell Program Synthesizer", Tech. Rep. 79-370, Dept. of Computer Science, Cornell Univ., 1979.

[10] Weinreb, D., and Moon, D., "Lisp Machine Manual", MIT AI Lab., March 1981.

Maclisp Compatibility

The discussion in the main body of this paper is couched in terms of Lisp Machine Lisp, however, GPRINT is substantially Maclisp compatible. Almost everything above applies equally to both versions. This section discusses the few differences between the two versions.

The I/O in Maclisp is quite different than on the Lisp Machine. The Maclisp version follows all of the Maclisp conventions. In particular, you can call GPRINT with a list of files and default output is controlled by the variables TYO, ^R, ^W, OUTFILES, etc.

The compilation environment is somewhat different in Maclisp. GPRINT must be loaded in in order for formatting functions to compile correctly because GF is a macro. On the Lisp Machine you don't have to take any special action in order for this to be the case when you are using GPRINT. In Maclisp you have to make sure that it is loaded into the compiler by a DECLARE in any file which defines formats. Also note that in Maclisp the functions which take optional control parameters (eg GPRINT, GPRINT1, GPRINC, GEXPLODE, and GEXPLODEC) are lexprs and need *LEXPR declarations.

In Maclisp, the functions triggered by TERMINAL STOP-OUTPUT and TERMINAL RESUME are triggered by typing control characters. The printer can be stopped by typing ^S. Printing can be resumed by typing ^C (^R in TOPS20 versions). Reprinting in full is triggered by ^P. In Maclisp these control characters are not set up by default. You have to call the function GSET-UP-PRINTER in order to get them defined. Note also that in Maclisp, the default symbol for depth abbreviation is "#" instead of "**".

The Maclisp version of GPRINT supports the formatting of hunks. Two basic mechanisms are supplied analogous to the ones described for arrays in the main body of the paper. If a hunk is a USRHUNK which takes messages (note that EXTENDs and the like are all USRHUNKs) then GPRINT checks the messages it accepts. If it takes the message :GFORMAT-SELF then GPRINT sends a :GFORMAT-SELF message with the object as argument to the object so that it can format itself. If a USRHUNK does not take a :GFORMAT-SELF message, but it does take a :PRINT-SELF or PRINT message then GPRINT treats the hunk as an atomic object and lets the standard printer print it. If a USRHUNK does not accept any of these messages, then it is treated as an ordinary hunk.

In order to format an ordinary hunk GPRINT first checks to see if there is a formatting function for the hunk. The user sets up a hunk formatter by adding a function to the list in the variable GHUNK-FORMATTERS. The purpose of this function is two fold: to test whether it is applicable to a hunk (in which case it returns T) and in this case to actually format the hunk. GPRINT calls each of these functions in turn passing it the hunk. As soon as one of them returns T it stops. If they all return NIL then the hunk is printed by default in the normal way (e.g. in parentheses with the CXRs separated by periods) in block format.

Functional Summary

This appendix describes all of the user functions supported by GPRINT.

GPRINT *object* **&optional** *stream format level length endline startline*

This is exactly analogous to PRINT except that it does pretty printing. The first argument is the object to be printed. The second argument specifies the stream to use for output. If it is missing then the standard system default is used (e.g. STANDARD-OUTPUT).

The third argument is a formatting function which defaults to NIL. If non-NIL it will be used by GDISPATCH to format the object. For example, (GPRINT FOO STANDARD-OUTPUT ':GFN-FORMAT) will use functional format for the top level of FOO no matter what the CAR of foo is. The last four arguments can be used to control abbreviation. They are used to set the values of PRINLEVEL, PRINLENGTH, PRINENDLINE, and PRINSTARTLINE respectively. If they are omitted, then the current bindings of these variables are used to control abbreviation.

GPRINT1 *object* **&optional** *stream format level length endline startline*

This is exactly like GPRINT except that it corresponds to PRIN1 instead of PRINT. (Unfortunately, the standard Maclisp grind package has already used up the name GPRIN1.)

GPRINC *object* **&optional** *stream format level length endline startline*

This is exactly like GPRINT except that it corresponds to PRINC instead of PRINT.

PL *object* **&optional** *stream format*

This is an abbreviation for (GPRINT *object file format* NIL NIL NIL NIL). It specifies that the object should be printed without abbreviation. It is quite handy at top level.

GFORMAT *stream template* **&rest** *args*

This is just like FORMAT except that GPRINT is called to do the printing and the *template* has the same form as a template for GF. For example, (GFORMAT NIL "(*_<*->)" X) creates a string containing X printed in functional format at the top level.

GEXPLODE *object* **&optional** *format level length*

This is analogous to the function EXPLODE except that it does pretty printing.

GEXPLODEC *object* **&optional** *format level length*

This is analogous to the function EXPLODEC except that it does pretty printing.

PLP &quote &rest *args*

This is very similar to GRINDEF but calls GPRINT. Each *arg* is either a symbol or a CONS of a symbol and a list of specific properties to print. If it is a symbol then any properties it has which are in the list PLP-PROPERTIES are printed. Otherwise, the specified properties are printed. If no *args* are supplied then PLP is reexecuted on the last set of args it was called on.

GSET-UP-PRINTER

Calling this sets up GPRINT as the top level printer. This consists basically of just setting the variable PRIN1 to GPRINT1.

GF *template* **&rest** *args*

This is used to define formatting functions. The structure of the *template* is summarized in a separate appendix. Note that unlike **GFORMAT** this does not actually print anything. Rather, it just makes queue entries when the formatting function it is in is called by **GDISPATCH**. The fact that **GF** is a macro saves time by parsing the template at compile time, and producing efficient code to do the formatting. This does waste space however. It is to your advantage to make each template as short as possible.

GFUNCTION *template*

This is an abbreviation for **#'(LAMBDA (X) (GF** *template* **X))**.

FORMAT *stream format-string &rest args*

A new format keyword **~N** is defined so that you can call **GPRINT1** from **FORMAT**. **~:N** invokes **GPRINC**. Numeric pre-arguments are taken to be **PRINLEVEL**, **PRINLENGTH**, etc.

Variable Summary

This appendix summarizes all of the control variables which can be set by the user in order to control the actions of GPRINT.

PRINLENGTH *system defined default*

This specifies the maximum length list that will be printed without abbreviation. NIL means infinity.

PRINLEVEL *system defined default*

This specifies the maximum depth at which any object will be printed. NIL means infinity.

PRINSTARTLINE *default* **NIL**

Output is inhibited until the PRINSTARTLINEth line is reached. NIL is the same as 0.

PRINENDLINE *default* **4**

Output is aborted and the printer returns normally as soon as the PRINENDLINEth line is reached.

PRINMARGIN *default* **NIL**

This specifies the total line length available for printing. If it is NIL, then the printer asks the output stream what the line length is.

MISER-WIDTH *default* **40**

Miser mode printout is triggered if there is less than this amount of width available for printing.

MAJOR-WIDTH *default* **40**

Left shifting of logical units will occur if there is less than this amount of width available for printing.

GCHECKRECURSION *default* **T**

If this is T then GPRINT checks for circular pointers and abbreviates them appropriately.

GSHOW-ERRORS *default* **NIL**

Normally, GPRINT does an ERRSET so that no error which occurs during formatting can cause an error in GPRINT. If this is set to T then you will enter the error handler if any error occurs. This is useful for debugging.

GFORCE-MORES *default* **T**

If this is T then things are set up so that you get MORE processing all of the time. Otherwise, MORE processing is suppressed if printing is initiated within 7 lines of the bottom of the screen.

GSPECIAL-FORMATTERS *default* **NIL**

This holds a list of formatting functions which are tested for applicability before any other dispatching is done.

GOVERRIDING-LIST-FORMATTERS *default* **NIL**

This holds a list of formatting functions which are tested for applicability to any list which is being printed before any other dispatching is done on it.

GLIST-FORMATTERS *default* **NIL**

This holds a list of formatting functions which are tested for applicability to any list which is being printed before any other dispatching is done on it *unless* dispatch was called with a specific suggesting of how to format the list. (The difference between this and GOVERRIDING-LIST-FORMATTERS is that these are applied in fewer places. For example, they will not be tested against the list of bound variables in a PROG because the format for PROG specifies exactly how this subpart of a PROG should be formatted.)

:GFORMAT *property*

If the CAR of a list has a value for this property, then the value is called as a formatting function to format the list. (If none of the above cases apply.)

GAPPLY-FORMAT *default* **:GAPPLY-FORMAT**

This is used as the format for literal LAMBDA applications.

GFN-FORMAT *default* **:GFN-FORMAT**

This is used as the default format for function applications.

GSYMBOL-CAR-FORMAT *default* **:G1TBLOCK**

This is used as the default format for lists whose CARs are symbols.

GNON-SYMBOL-CAR-FORMAT *default* **:G1TBLOCK**

This is used as the default format for lists whose CARs are not symbols.

:GFORMAT-SELF *message*

If an instance, entity, or named-structure is set up so that it will process this message type, then it is sent a message in order to format itself. It gets one argument (the object itself) in addition to any arguments which are supplied by the message sending mechanism.

GARRAY-FORMATTERS *default* **NIL**

This holds a list of formatting functions which will be tested for applicability to any array being printed which doesn't take a :GFORMAT-SELF message.

GHUNK-FORMATTERS *default* **NIL**

This holds a list of formatting functions which will be tested for applicability to any hunk being printed which doesn't take a :GFORMAT-SELF message.

GRIND-MACROEXPANDED *default* **NIL**

If this is T then MACROMEMOized macros will printed out as they appear after expansion. Otherwise they will be printed out as they appear before expansion.

PLP-PROPERTIES *default* **(:FUNCTION :VALUE)**

This holds the list of values which the function PLP will print out by default. The default specifies that only the function value and value should be printed.

Summary of Formatting Codes

This appendix summarizes the formatting codes which are available for use in the template supplied to the macro GF. The template is a string of single character commands, some of which can be followed by a parameter. There are three kinds of parameters:

n - Some commands take a number as a parameter. This number should be an integer optionally beginning with a "-" and/or ending with a ".". Alternately, it can be omitted in which case a default value is used.

f - Some commands take a function name as a parameter. This name is an arbitrary symbol possibly containing ":". Case does not matter. The symbol must be terminated by a blank. Function name parameters cannot be omitted. They have no default values.

- This can be used in place of any numeric parameter or any function name parameter. It indicates that the next input to GF should be used as the parameter, instead of a literal value.

The commands which can be used in a template are divided into several categories. The first set is used to parse the structure of the arguments to GF so that their parts can be accessed.

[ ] - This is used to access the internal elements of an item which is a list. The template inside the brackets refers to the elements of the list. If the item is not a list, then no formatting of it, or anything inside it, is done. Processing begins by considering each element of this list in turn. As soon as the list is exhausted, control skips out of the subtemplate and continues after its end. This is done even if there is more stuff left in the subtemplate. Special code is included to deal with the possibility of unexpectedly encountering a non-NIL atomic CDR. If this happens it is *automatically* formatted to appear after a ".". [] also produces special code to deal with length abbreviation. They only way to get it automatically is to use [].

. - (Period) This is valid only inside []. It specifies that the next item is the whole sublist left to process by [] rather than its CAR. For example, (GF "[*_.*]" '(1 2)) is the same as (GF "*_*" 1 '(2)).

Note that when a "." is used, normal checking for the end of the list in the [] is suppressed. For example, (GF "[*_.*_]" '(1)) is equivalent to (GF "*_*_" 1 NIL). The NIL at the end of the list is explicitly picked up by the ".", and a blank will be printed at the end. This happens even though the [] template would normally have terminated right after the first *.

< > - This can only be used directly inside [] (or ()). It specifies an indefinite repeat block. This is used to specify a template for a list of unknown length.

The next set of commands are used to specify how individual items are printed out.

I - Ignore the corresponding item.

'*literal*' - Print the indicated literal using PRINC and do not count it as one of the items printed from the point of view of length abbreviation. Note that in the *literal* "'" stands for "'".

* - This specifies that GDISPATCH should be called in order to format the corresponding item.

%*f* - This specifies that the function *f* should be called in order to format the corresponding item. (Note if *f* is # then the argument which is used as the function follows the argument which is formatted.)

$*f* - (Dollar sign) This command specifies that GDISPATCH should be called in order to format the corresponding item, but that the function *f* should be passed to GDISPATCH as a suggestion of how to format the item. (Note if *f* is # then the argument which is used as the function follows the argument which is formatted.) The difference between $*f* and %*f* is that with $*f* GDISPATCH gets control. As a result, if the item is not a list, or if some function on GOVERRIDING-LIST-FORMATTERS formats it, then the function *f* will not get used.

$/"*subtemplate*/" - In addition to the name of a function, the parameter to $ can be a literal template which is converted into a function to use. (Note that the quotes have to be slashified in order to read in inside a quoted string.) The formatting function produced is compiled out of line. As a result, if there is a # format code in it, the argument to GF that this refers to will be compiled out of line. In order for this to work any variables this refers to must be declared special.

The next commands are used to specify the nested structure of the output (which need not be the same as that of the input).

{*n*) - This indicates a substructural unit in the output. The parameter specifies what indentation to use when printing out the items inside the substructure if the substructure cannot be printed on a single line. (If the indentation is specified to be zero then the substructure is not counted as increasing the depth from the point of view of depth abbreviation.) The default parameter value is calculated as the sum of the lengths of the first thing printed in the substructure, and any literals before it and any spaces after it.

+*n* - (Plus) This specifies a change in indentation. The indentation level in the current substructure is incremented by *n* which can be negative. Note that this will not take effect until the next line. For example, the template "(*-*-+2*-*)" does not increase the indentation until the fourth item is printed while "(*-*+2-*-*)" prints the third item at an increased indentation.

(*n*) - This is a useful abbreviation in the situation where the nested structure of the output is the same as the nested structure of the input, and when you want to print parentheses around the structure. It is an abbreviation for {*n*'('[ ]')'}. Additionally, if the (*n*) is nested more directly inside [] than inside $ then it is treated as an abbreviation for $/"{*n*'('[ ]')'}/". In other words, if the item whose format is being specified by the (*n*) was not passed through GDISPATCH for dispatching then the $ format code is used to force the list to dispatch through GDISPATCH. This prevents the format from blowing up when the item is not a list. (Note the comment about # inside $/" /" above.)

The next set of commands specifies spacing and where and when carriage returns should be printed. Note that there is actually a complete separation between these two concepts. The format codes used above which combine the two ideas are abbreviations combining the underlying codes.

~*n* - (Tilde) Print *n* (default 1) spaces. (Note that spaces are elided if they are the first or last thing on a line).

T*n* - Tab over. Moves to a place where the character position relative to the current indentation is congruent to zero modulo *n*. (Does not move at all if it does not have to.) When necessary, a default tab size is calculated based on the length of the other items in the substructure.

A - Do a line break here always.

I - Same as A.

N - Do a line break here if required for normal mode printing. I.e. if and only if the structure immediately containing this point cannot be printed on a single line.

-*n* - (Minus) Abbreviation for "~*n*N" which is what you usually want.

B - Do a line break here if required for block mode printing. This is the same as N except that even if the immediately containing structure is being broken up a line break will not be put here as long as the following structure can be printed on the end of the current line and the prior structure at this level was printed on a single line.

,*n* - (Comma) Abbreviation for "~*n*B" which is what you usually want.

;*n* - (Semicolon) Abbreviation for "~1T*n*B" which is what you often want.

M - Do a line break here if required for miser mode printing. Put a line break here if the containing structure will not fit on a single line, and the remaining line width available for printing is less than MISER-WIDTH.

_*n* - (Underscore) Abbreviation for "~*n*M" which is what you usually want.

The next two formatting codes were not discussed above. They are provided as extra hooks into the GPRINTing process.

&*f* - The function *f* is called with no arguments at this point. Note that function is called during the formatting process.

E - When the output routine gets to this point in printing, the arg to GF corresponding to the E is EVALed (out of line). This is useful for getting information about the state of the printing process. It should NOT be used to print anything out because the output routine will not realize that anything was printed and its character position calculations will be wrong. Note that the difference between & and E is the time at which the function evaluation occurs.

The characters SPACE, TAB, CR, and LF are all ignored. Any other character is an error.


ATE
LME